Reuse-Oriented Reverse Engineering of Functional Components from X86 Binaries

Dohyeong Kim[1] William N. Sumner[2] Xiangyu Zhang[1] Dongyan Xu[1] Hira Agrawal[3]
[1]Department of Computer Science, Purdue University
[2]School of Computing Science, Simon Fraser University
[3]Applied Communication Sciences
[1]{kim1051,xyzhang,dxu}@cs.purdue.edu [2]wsumner@sfu.ca
[3]hagrawal@appcommsci.com

ABSTRACT

Locating, extracting, and reusing the implementation of a feature within an existing binary program is challenging. This paper proposes a novel algorithm to identify modular functions corresponding to such features and to provide usable interfaces for the extracted functions. We provide a way to represent a desired feature with two executions that both execute the feature but with different inputs. Instead of reverse engineering the interface of a function, we wrap the existing interface and provide a simpler and more intuitive interface for the function through *concretization* and *redirection*. Experiments show that our technique can be applied to extract varied features from several real world applications including a malicious application.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Algorithms

Keywords

Feature Extraction, Reverse Engineering, Dynamic Analysis

1. INTRODUCTION

Developers have long struggled with the desire to reuse previously implemented features within new code. By reusing an old implementation, developers can avoid creating new bugs and can create easier to maintain programs [32, 26, 19, 8]. Implementation reuse can also be crucial when a new program must replicate features within a legacy system, but the specification for the legacy system no longer exists. Driven by this desire to reuse existing implementations, previous research has delved into techniques for both *locating* the implementation of a feature within a body of source code [40, 41, 8, 10, 15, 33, 9, 38, 35, 13, 6, 14, 16] and *extracting* that source implementation into a conveniently reusable function [26, 23, 17, 18, 31, 25, 10].

These techniques generally assume the availability of the original source code, but in practice the source code itself may no longer be available or may no longer even exist. Indeed, facing the maintenance of legacy programs without source code, DARPA recently called for a solution to this exact problem [3]. For example, some components are provided to developers only in the binary form, and the source of these programs or libraries may not be available due to intellectual property restrictions [36]. In other cases, companies may have existing programs that implement *de facto* specifications, but both the original source code and any documentation of the specifications have been lost over time [3]. Finally, when reverse engineering the behavior of a foreign program, a program from a third party, security researchers sometimes wish to extract certain features, such as encoding/decoding routines [22] or anti-debugger techniques [20] from a foreign program in binary form. Reusing these features from foreign binaries allows the security analysts to gain insight into the behavior of malicious code and potentially develop defensive techniques [22]. In each of these scenarios, a developer needs to locate and extract the existing implementation of a feature that exists only in binary form within an existing program. Although source code provides rich information about the behavior and structure of a program, much of this information is stripped away when the program is compiled to a binary form.Thus, techniques for locating and extracting features that rely on source code analysis and manipulation no longer apply.

New solutions must be found for both locating and extracting features. Prior work on locating a desired feature includes techniques built on statement coverage information [41] and dynamic slicing techniques [43]. While both of these techniques apply to both source code and binaries, they may both locate features too coarsely, including more of the original implementation than is necessary or desirable. Dynamic slicing techniques compute transitive closures over the dynamic dependence graph of an execution [24, 5, 46]. These closures are known to be large in practice [45]. Statement coverage techniques contrast the statements performed within an execution that exhibits a desired feature against those performed in an execution that does not exhibit the

ICSE '14, May 31 - June 7, 2014, Hyderabad, India

feature. The intuition is that statements executed only, or more frequently, within the execution exhibiting the feature should implement the feature itself. We observe and later show that such approaches can be too coarse grained and identify portions of the original implementation that are unnecessary for implementing the feature.

Once the functions implementing a feature have been identified, they may be extracted from the original binary, but extraction alone is insufficient. To reuse the extracted component, we must provide an interface through which it may be invoked, but even state of the art binary analysis tools have difficulty reverse engineering such interfaces. We show that the original interface for a component can also involve complex heap structures, and the parameters that correspond to a feature of interest may be deeply embedded within these heap structures and subject to subtle constraints. Developers should not need to deal with such complexities when reusing an extracted component.

In this paper, we propose a novel approach to locating modular functions that correspond to a desired feature and providing a usable interface for the extracted component. We describe the desired feature through multiple executions that use the feature. The user executes the feature twice with different inputs, and our technique semantically contrasts the executions to discover where the different input values are used and which part of the code produces the desired output from the input. Our technique then isolates the feature using *concretization* to replace the original parameters of the function with values from a real run. Finally, it wraps the original interface of the binary with a new and simpler interface for the developer to invoke and then uses *redirection* to ensure that the parameters of the new interface are used consistently throughout the extracted function.

Our main contributions are highlighted as follows.

- We provide a way to precisely represent a desired feature using multiple executions that exhibit the feature. Providing these executions is intuitive to a user who knows how to use the existing binary.
- We perform a semantic comparison of the provided executions using dual slicing [39]. This allows us to precisely locate the desired feature within the code.
- We propose a technique called *interface casting* that uses *concretization* to isolate desired parameters for a function. It then wraps an extracted binary feature with an adapter and exploits *redirection* to use the parameters of the adapter. This provides the developer a convenient means of invoking the extracted feature.
- We implement and evaluate a prototype of the approach. We apply our technique to 8 applications and extract 10 reusable components from the binaries. We show that even when there are originally no parameters to the extracted functions, our technique still applies.

2. MOTIVATING EXAMPLE

Suppose a developer desires to extract and reuse the email sending feature of `pine`, an email client. Because `pine` has many diverse email features, this *reuser* must first locate the function that contains the desired feature. To reuse the function, they must also uncover the function's interface or prototype. Knowing the interface, they can provide parameters such as the sender address, recipient, subject, and body for a sent email. In this section, we show how to locate the desired function by using dual slicing. We then show how to extract the function into an isolated component with a reusable interface by using concretization and redirection.

2.1 Function Location

Before extraction, we must locate the function responsible for sending an email. To find the function, we contrast two different executions of `pine`, each of which sends a different email. We follow the same steps both times, except that the sender addresses, recipients, subjects, and bodies of the sent emails differ. Thus, we choose the same menu items in the same order, and we provide the same sequence of key strokes except for the four parameters of interest. As a result, the two executions follow the same paths through the program except for differences related to the differing user input.

Dual slicing is a technique that contrasts two executions and identifies only those instructions that both behave differently across the two executions and contribute to their different outputs. Intuitively, the user inputs for the two executions of `pine` differ only with respect to the emails sent, so the two executions should mainly differ in the portion of code that is responsible for processing and sending the different emails. Thus, the differences identified by the dual slice should be the behaviors of `pine` that we wish to extract.

Next, our technique find the function that encloses all of the relevant differences between the two executions. Fig. 1 shows the part of the dynamic call tree containing the dual slice. Each node represents a function and each arrow represents a caller-callee relationship between functions. Shaded nodes represent those functions containing the relevant instructions within the dual slice. The topmost shaded function is `call_mailer()`, and it transitively calls all other shaded functions. Since the shaded functions are necessary for the mail sending component, we identify `call_mailer()` and its callees as the components of `pine` to extract.

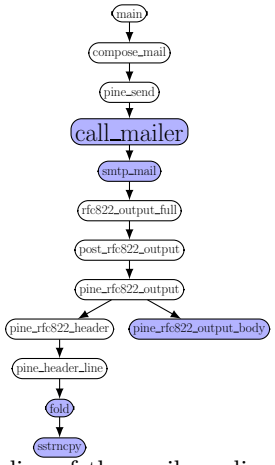


Figure 1: Dual slice of the mail sending feature in `pine`. Shaded nodes show the dual slice within the call tree.

2.2 Interface Casting

We must next extract `call_mailer()` and provide it with a usable interface. The new interface is particularly important because the original interface is complex and does not match the expectations of reuser.

Suppose that we tried to invoke the extracted function directly. Fig. 2 presents the original interface of `call_mailer()`. The function has 6 parameters. The first and the second

```
struct BODY {
  PARTTEXT contents;     /* body part contents */
  union {   /* different ways of accessing contents */
    PART *part;   /* body part list */
    MESSAGE *msg; /* body encapsulated message */
  } nested;
  ...
};
struct PART {
  BODY body;  /* body information for this part */
  PART *next; /* next body part */
};
struct MESSAGE {
  BODY *body; /* message body */
  PARTTEXT text /* body text */
  ...
};
struct PARTTEXT {
  unsigned long offset;  /* offset from body origin */
  struct {
    unsigned char *data; /* text */
    unsigned long size;  /* size of text in octets */
  } text;
}
int
call_mailer(METAENV *header, BODY *body,
  char **alt_smtp_servers, int flags,
  void (*bigresult_f)(char *, int),
  void (*pipecb_f)(PIPE_S *, int, void *))
```

Figure 2: Interface of `call_mailer()`

arguments are pointers to internal data structures, and we would need to reverse engineer those data structures to reuse the original interface. In particular, the body of an email is stored in `body->contents.text.data` and the size of the body is stored in `body->contents.text.size`. To specify the body, we would first need to allocate memory regions for the `BODY` structure and its child data structures, e.g. `PART` and `MESSAGE`, and specify correct values for both `data` and `size`. This also requires understanding the semantic relationship between `data` and `size`. To reuse `call_mailer()`, we would further need to correctly initialize the entire data structure and identify the semantics of each field *even if the field is unrelated to the four parameters we want to provide*. In fact, there is even more complexity, as email contents may be specified in two different ways: one through the `contents` field of `BODY` and the other through a further nested field of `BODY`. Expecting the reuser to manage this complexity on their own is unrealistic.

To provide a usable interface for the function, we must simplify away the unnecessary parameters and introduce new parameters matching the reuser's intentions. We call this process *interface casting*. To simplify existing parameters, our technique first statically concretizes those values generated outside `call_mailer()` and used inside `call_mailer()`. Thus, if `call_mailer()` is invoked by the reuser, the parameter `bigresult_f` does not actually take a variable argument. Instead, we provide it a concrete value observed in one of the original executions. We concretize not only the values of all function parameters but any memory values defined outside `call_mailer()`. By concretizing all of the direct and indirect inputs, we hermetically seal the function of interest. That is, by providing concrete values for all inputs of a function, we ensure that it behaves the same way every time.

To provide the parameters desired by the reuser, our technique then relaxes this seal to allow only the chosen parameters to again affect the function's behavior. We redirect accesses of the original inputs to use memory locations for the new parameters provided by an interface that we construct. Since we already concretize the parameters and memory values, the location for a parameter is statically fixed. For example, the value of the parameter `body` may be concretized to `0x0408CC00`, and the subject of an email may be concretized to `0x0409DB00`. Thus, the program will always read the email subject from the same location in memory, and we can redirect accesses of that memory location to use a new memory location that contains a new subject string.

Before we can relax and redirect accesses of inputs, we must first allow the reuser to determine just which data should be parameters for the extracted function. Once again, the reuser can use dual slicing to provide this information. Recall that in the function identification phase we contrasted two executions with all desired parameters changed to identify the code to extract. In contrast, to identify the instructions that read each input, we need only change one input at a time. This way the dual slice between the original execution and the execution with one differing input will capture only those instructions processing the changed input.

Once we identify all desired parameters of `call_mailer()` and create a new interface using concretization and redirection, we can simply extract the function from its original binary by using binary rewriting tools[11]. The new interface we provide acts as a wrapper that invokes this binary function, allowing the reuser to call it like any other library function.

3. THE REUSE PROCESS

In this section we discuss the details of reusing functions from binary code. We present algorithms for both locating the function that contains a feature through dual slicing and for providing a reusable interface through concretization and redirection.

3.1 Component Location

We first present background information on dual slicing to clarify details of our approach for locating components within a binary. We then explore our algorithms for locating components and why they can localize a component to a more concise portion of code than existing techniques.

Dual slicing. Dual slicing is a slicing technique that contrasts two executions and produces a slice containing only those differences between the two executions that are responsible for some observably different behavior [39]. Alg. 1 presents the core algorithm. Given a slice criterion (e_1, e_2) that identifies some output differences across the two executions, the algorithm computes a set of dynamic dependences from both executions. (e_1, e_2) denotes that two execution points e_1 and e_2, in the first and the second executions respectively, align or correspond across the executions [44]. $(e_1, \perp)$ denotes that there is no execution point in the second execution that aligns with e_1 in the first execution.

The algorithm first ensures that the slice criterion exists in the first execution. Lines 2-6 process data dependences at the slice criterion. Here, $\{(e_1, e_2) \longrightarrow (e_1', e_2')\}$ denotes that e_1 has a data dependence upon e_1' in the first execution, e_2 has a data dependence upon e_2' in the second execution, and e_1' aligns with e_2'. e_2' can be $\perp$ when e_1' does not align with any point in the second execution or e_2 is not data dependent on the alignment of e_1'. On line 3, if the data dependence exists only in the first execution or if the values of two data dependences differ, the data dependence is added to the dual slice. The algorithm proceeds to include the dual slice from (e_1', e_2') recursively, similar to traditional dynamic slicing. Lines 7-10 process the control dependence of the slice criterion, denoted as $\Longrightarrow$. Similar to the data dependence,

if the control dependence exists only in the first execution or the branch outcomes differ, the control dependence and the recursive dual slice of the control dependence are added to the dual slice. So far, the algorithm considers only data dependences and control dependences when e_1 is not null. Lines 12-14 compute the dual slice when e_2 is not null.

Algorithm 1 Dual Slicing

Input: e_1, e_2- slice criteria
Output:$\mathcal{D}$- the dual slice, a set of deps in either execution

```
DUALSLICE(e1, e2)
 1: if e1 ≠ ⊥ then
 2:   for all data dep dd ← {(e1, e2) ⟶ (e1', e2')} do
 3:     if e2' ≡ ⊥ or values at e1' and e2' differ then
 4:       D ← D ∪ dd ∪ DUALSLICE(e1', e2')
 5:     end if
 6:   end for
 7:   control dep cd ← {(e1, e2) ⟹ (e1', e2')}
 8:   if e2' ≡ ⊥ or branch outcomes at e1' and e2' differ
      then
 9:     D ← D ∪ cd ∪ DUALSLICE(e1', e2')
10:   end if
11: end if
12: if e2 ≠ ⊥ then
13:   /* operations symmetric to when e1 ≠ ⊥ */
14: end if
15: return D
```

Component location using dual slicing. To use dual slicing to identify the component that corresponds to the desired feature, we use two executions that each exercise the desired feature but that use different inputs. For example, in the `pine` case study, we send an email in both executions, but the emails have different recipients, subjects, and bodies. The resulting dual slice contains only those instructions that process the input because all the execution differences originate from the differing inputs. The slice also includes only instructions that help produce the desired output because slicing excludes instructions unrelated to the output.

Alg. 2 presents the component identification algorithm. The algorithm requires two executions, E_1 and E_2, which exercise the same feature but with different inputs. Lines 1 and 2 choose the output corresponding to the desired feature as the slice criterion. In the `pine` example, we use the network packet containing the composed email as the slice criterion. Line 3 computes the dual slice, and line 4 trims off a prefix of the slice that only moves the arguments around without using them for any computation. Line 5 locates the function that contains this trimmed dual slice. It chooses the closest common ancestor function in the dynamic call tree of those functions whose instructions reside in this dual slice. Line 6 further selects this common ancestor as well as all functions that it transitively called in E_1 and E_2 as targets of extraction. In other words, the identified components comprise nodes of the dynamic call tree with the identified common ancestor function as their root.

Suppose that we wish to identify the function containing the 'email sending' functionality of the sample program in Fig. 3 that models `pine`. `load_config()` first initializes global variables that will be used in `pine_send()`. `menu()` waits for an input from a user with timer of 1 second. If the timer expires, `menu()` performs background tasks. If the user instead selects

Algorithm 2 Component Location

Input: a pair of executions E_1 and E_2 with both exercising the target functionality but with different inputs.

```
IDENTIFICATION(E1,E2)
1: (O1,O2) = outputs corresponding to the desired feature
   in E1 and E2, resp.
2: (e1, e2) = execution points that emit O1 and O2, resp.
3: ds_e = DUALSLICE(e1, e2)
4: ts_e = TRIM(ds_e)
5: func = the modular function that encloses ts_e
6: extract = func and all user functions directly/indirectly
   called by func
7: return extract
```

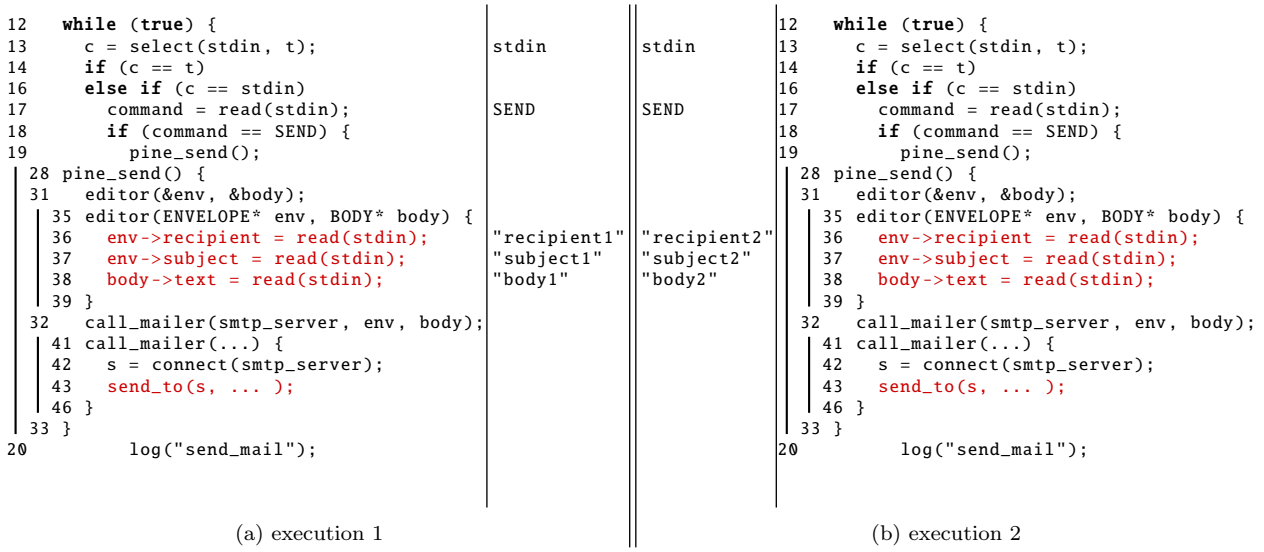
the send menu option, `pine_send()` calls `editor()` to edit the recipient, subject, and body of an email. Later `call_mailer()` composes an email with the information from `editor()` and sends it to an SMTP server. In this example, `call_mailer()` has the email sending functionality since `editor()` only stores the user input into a buffer and does not apply any calculation or transformation to the given inputs.

To get two execution traces for dual slicing, we run the program twice with different recipients, subjects, and body texts. We run the program in exactly the same way the second time except for those inputs. Those three parameters are all that we want our extracted component to require, and we want to use the same values for other configurations such as the SMTP server address and sender address.

Fig. 4a and Fig. 4b present the resulting traces. Line 43 is the slice criterion because it sends the packet containing the email to the SMTP server. The dual slice includes that line because the sent packets differ in the two executions. Line 43 further depends on lines 42, 36, 37, and 38. Line 42 uses the same value of `smtp_server` in both executions, so the dual slice excludes it. Because lines 36, 37, and 38 produce value differences, the dual slice includes them. Also, line 43 is (directly or transitively) control dependent upon lines 12, 14, 16, and 18, but those lines do not reflect differences across the two executions, so the dual slice excludes them.

The dual slice shows that lines 36, 37, 38, and 43 are important for the mail sending functionality. Note, however, that lines 36, 37, and 38 simply copy the input to a buffer. Because they only move the input around and do not make decisions or perform computations with it, these lines form an *irrelevant prefix* of the desired behavior. They reflect preparatory bookkeeping work rather than behavior of the desired component itself. We can thus omit them entirely and still locate the functions containing the behavior we wish to extract. The TRIM function removes such instructions from the front of the dual slice up until the first decisions or computations with inputs that differ across the executions. In practice, this localizes the component to a smaller portion of code. In our example, the only remaining instruction is line 43, so the technique identifies that the email sending feature is located within `call_mailer()`.

Coverage based approaches. Prior work on feature location computed the difference in statement coverage between two executions: one sending an email and the other not sending the email [41]. This coverage based comparison can identify more functions than we desire because the reuser cannot control the program's behavior at a fine-grained level.



(a) execution 1 (b) execution 2

Figure 4: Dual slice of the simplified `pine` example from Fig. 3.

That is, a small behavioral difference to the user may correspond to many differences in terms of which functions a program executes, only a few of which may be interesting.

Consider **xv**, an image viewer that can convert one image format to another. Suppose our target functionality is converting a BMP format image to JPEG format. To compute the coverage difference, we load the same file in both executions. We convert the file into JPEG format in one execution but cancel the conversion in the other. Fig. 5a presents the coverage comparison results. The results show a large call graph with many functions related to processing the user interface and handling user input such as mouse clicks in addition to the important function, `writeJPEG()`. Furthermore, the approach misses the function `LoadBMP()`, which is responsible for loading a BMP image. In contrast, dual slicing computes a concise set of functions for the conversion and identifies `LoadBMP()` as well. Fig. 5b shows the dual slice, which highlights only the important functions: `writeJPEG()` and `LoadBMP()`. We later discuss pruning the extracted component to contain only these two functions and their callees.

The simple coverage difference includes many non-essential functions because the reuser cannot control every detail of program behavior by enabling and disabling the target feature. Hence, in this paper we use dual slicing[39] to focus more concisely on the interesting differences.

3.2 Interface Casting

Once we have located the function the reuser wishes to extract, we must take the potentially complicated interface of that original function and compose a simpler alternative interface with only the reuser's desired parameters. Our approach to this problem is to first concretizing all of the values that feed into the selected function to hermetically seal and isolate the function's behavior. This makes the function behave the same way every time it executes. Our technique then relaxes this seal for only the reuser's desired inputs by redirecting accesses of the original inputs so that they instead access inputs of a freshly constructed interface.

Alg. 3 presents an overview of the interface casting process. The algorithm takes three parameters: (1) the code to extract, a result of the component identification algorithm, (2) an execution E that exercises the desired feature, and (3) one additional execution for each parameter we wish to specify. In the `pine` case study, if we wished to specify the recipient, subject, and body of an email, we would need three additional executions. One would send an email with the same subject and body as execution E but with a different recipient. Another would send the email with a different subject. The last would send the email with a different body. These additional executions identify those instructions that access each of the different specified parameters.

Algorithm 3 Interface Casting

Input: *extract* denotes the code to extract, which is identified in the previous section; an execution E exercising the target functionality; a list S of pairs (E_i, T_i) with E_i the same as E except that E_i has a different value for the ith input (with type T_i) intended by the user.

INTERFACECASTING(*extract*,E,S)
1: $\overline{ext\_dep}$= instruction instances in E that are part of *extract* and have external dependences
`/* Seal off all external dependences */`
2: CONCRETIZE(*extract*, $\overline{ext\_dep}$)
`/* Patch to allow reuser specified inputs */`
3: **for each** $(E_i, T_i) \in S$ **do**
4: (e, e_i)= instruction instances emitting the feature related output in E and E_i, resp.
5: $\overline{diff}$= E's instruction instances in DUALSLICE(e, e_i)
6: $\overline{if}$ = instruction instances in $\overline{diff} \cap \overline{ext\_dep}$
7: REDIRECT(*extract*, $\overline{if}$, T_i)
8: **end for**

Line 1 computes the set of instruction instances with *external* dependences. If an instruction reads a value from memory that was written outside the modular component, it is an external dependence. Since the selected component does not create values for external dependences, they must be provided for the component to execute correctly. Line 2 concretizes all external memory dependences to seal the behavior of the function. This replaces values of accesses with those observed in E. Extracting the function at this point would create a new function with no arguments that behaves as in E every time it is called. The loop in lines 3-8 consid-

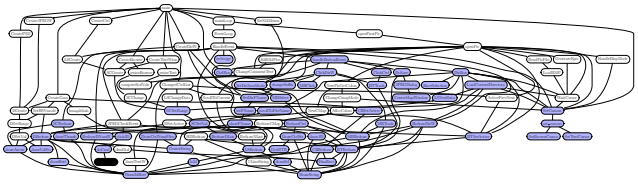
```
1  main() {
2    load_config();
3    menu();
4  }
5
6  load_config() {
7    smtp_server = x.x.x.x;
8  }
9
10 menu() {
11   t = timer(1);
12   while (true) {
13     c = select(stdin, t);
14     if (c == t) // timer expired
15       do_something();
16     else if (c == stdin) {
17       command = read(stdin);
18       if (command == SEND) {
19         pine_send();
20         log("send mail");
21       }
22       else if (command == CANCEL)
23         continue;
24     }
25   }
26 }
27
28 pine_send() {
29   ENVELOPE env;
30   BODY body;
31   editor(&env, &body);
32   call_mailer(smtp_server, env, body);
33 }
34
35 editor(ENVELOPE* env, BODY* body) {
36   env->recipient = read();
37   env->subject = read();
38   body->text = read();
39 }
40
41 call_mailer(char* server, ENVELOPE* e, BODY* b) {
42   s = connect(server);
43   send_to(s,
44           compose_mail(e->recipient,
45                        e->subject, b->text));
46 }
```

Figure 3: A program modeling 'email sending' in `pine`.

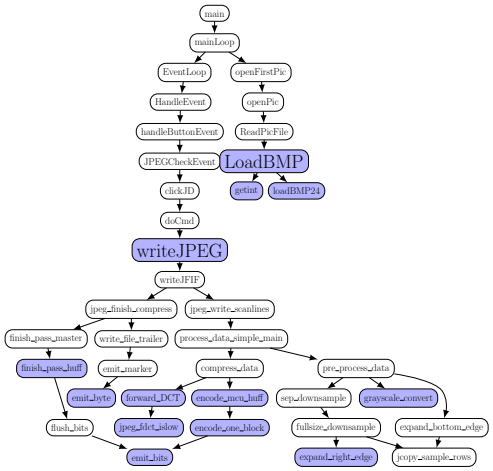
ers each parameter specified by the reuser and identifies all instructions with external dependences upon each parameter. The loop redirects those accesses to instead use new memory locations that hold the values of the parameters within a wrapper function that matches the reuser's demands.

Concretization. To seal the function and remove undesired inputs from the interface, we concretize the values of those inputs by monitoring memory accesses. For example, in Fig. 2, `call_mailer()` has a variable `alt_smtp_servers` that holds alternative SMTP server addresses. We do not want the function we extract to expose this complex behavior to the reuser. To provide an interface without this parameter, we concretize the value of the parameter, so `alt_smtp_servers` always holds the same value in the extracted version of the function. Note that if the reuser intends the SMTP server to be an input of the extracted component, he/she could simply provide an additional execution that differs from the original execution only at the SMTP server address.

Alg. 4 explains the concretization process. Lines 1-5 process the instructions with external dependences. By the definition of an external dependence, we can assume that the instruction i will have the form `MOV` r_2`, [`r_1`]` because it reads external memory. Lines 3-4 replace the original instruction with (1) a guard to see if the dynamic instance of the instruction uses external memory and (2) new `MOV` instructions to redirect the memory access to a saved value if so. Lines 6-13 process the instructions that write to external



(a) Coverage difference between converting BMP to JPEG and not converting BMP to JPEG. Shaded nodes are the functions executed only when converting BMP to JPEG and the lone black node is `writeJPEG`, which is important for the conversion.



(b) The result of dual slicing. The call graph is concise and highlights only two functions, `LoadBMP()` and `writeJPEG()`.

Figure 5: Call graph from `xv` case study

memory. In other words, such an instruction writes a value to some location in memory allocated outside the identified function in the original execution E, implying that the address is invalid in the extracted binary. Thus, we first map the observed memory address of the access into the address of a new variable that we create within the data section of the extracted binary on lines 9-10. Line 12, similar to the read case, replaces the instruction to use this new mapped address instead of the original one.

Redirection. In order to redirect parameters, we first identify the parameters using dual slicing. The approach is similar to the one used for locating the desired function. For each parameter, we use two inputs that differ only with respect to that parameter. For example, we may use two inputs that have different recipients to identify the instructions responsible for the recipient parameter.

After our technique identifies these parameter providing instructions, it redirects the memory accesses in the instructions to new locations. When an instruction reads a parameter from memory, it instead redirects the memory access to a new variable or buffer prepared to hold the parameter.

Alg. 5 presents the redirection algorithm. Line 1 adds a new variable to the data section of the binary. This new variable will hold the input for the extracted version of the function, so accesses of the original data must be redirected to this new variable. We break the inputs down into two different categories during the process: (1) *scalar* variables, which are always accessed through their starting address, and (2) *buffer* variables, which have many internal addresses that may be accessed independently. Lines 2-7 process scalar variables. On line 3, the algorithm iterates over the instructions $\overline{if}_i$ discovered by dual slicing with two inputs that identify one

Algorithm 4 Patch the extracted code for external dependences through concretization

Input: *extract* denotes the code to extract; $\overline{ext\_dep}$ denoting instruction instances in E that are part of *extract* and have external dependence

CONCRETIZE(*extract*, $\overline{ext\_dep}$)

```
   /* concretize reads of external dependences */
 1: for each unique instruction i in ext_dep do
 2:   let i be 'MOV r2,[r1]'
 3:   let T = {addr ↦ val} be a map from addresses to
      the values of i's external dependences
 4:   replace i with the following:
        if r1 ∈ T:
          MOV r2,T[r1]
        else: MOV r2,[r1]
 5: end for
   /* patch instructions writing through addresses
   derived from external dependences */
 6: for each instruction i in extract that may write to an
    address that is directly/indirectly computed from an
    external dependence in E do
 7:   let i be 'MOV [r2],r1'
 8:   for each external address a that i has written to
      do
 9:     add an entry x_a to the data section
10:     map[a] = x_a
11:   end for
12:   replace i with the following:
        if r2 ∈ map:
          MOV [map[r2]],r1
        else: MOV [r2],r1
13: end for
```

Algorithm 5 Redirect instructions related to the ith input.

Input: *extract* denotes the code to extract; $\overline{if}_i$ the instructions load the ith input; T_i the type of the ith input

REDIRECT(*extract*, $\overline{if}_i$, T_i)

```
 1: add a global variable v_i of type T_i to the data section
 2: if T_i is a scalar type then
 3:   for each unique instruction x in if_i do
 4:     let x be 'MOV r2,[r1]'
 5:     let addr be the address accessed by x in if_i
 6:     replace x with the following:
          if r1 == addr:
            r1 = &x_i
            MOV r2,[r1]
            r1 = addr
          else: MOV r2,[r1]
 7:   end for
 8: else if T_i is a buffer type then
 9:   for each unique buffer access instruction x in if_i
      do
        /* x must be an instruction repetitively
        executed to access a buffer */
10:     let x be 'MOV r2,[r1]'
11:     let addr be the lowest address accessed by x in
        if_i
12:     replace x with the following:
          if addr ≤ r1 < addr + T_i.size:
            t = r1
            r1 = &x_i + (r1 - addr)
            MOV r2,[r1]
            r1 = t
          else: MOV r2,[r1]
13:   end for
14: end if
```

parameter. On line 6, it replaces the instruction. If the instruction reads from the location that we identified as the ith parameter, it is redirected to instead read the new variable prepared for the ith parameter on line 1. External dependences through registers are handled similarly and thus elided. Lines 8-14 process a variable holding a buffer that may be read from at any consecutive memory locations within the buffer. Many strings provided by the user fall into this category, including the body `data` parameter in the `pine` case study. Similar to scalar variables, the algorithm iterates over and replaces the discovered instructions. If the instruction reads from a memory address corresponding to the ith parameter, it is redirected to read from the new location. Because the instruction reads from a buffer, we must guard the redirection by checking whether the address lies between the lowest address observed for the ith parameter and the size of the new parameter.

Next we use the `pine` example to illustrate concretization and redirection. Fig. 6 presents a portion of code from `pine` that reads the email subject along with the corresponding binary code and the rewritten binary after concretization and redirection. On line 3, `call_mailer` reads `header->env`, on line 14 `smtp_mail` reads `env->subject`, and on line 23 `rfc822_output_...()` reads a subject from a buffer where `subject` is pointing.

To find the instructions that read the email subject in this example, we slice two executions with different subjects. The resulting dual slice includes only line 23 because that instruction reads the subject, and the value changes when the user input changes. Although lines 3 and 14 are not in the dual slice, the values they use come from outside `call_mailer()`, causing external dependences. Hence, our technique automatically concretizes accesses in lines 3 and 14 and redirects the one in line 23, as shown in lines 5 (for `header`), 7 (for `header->env`), 16 (`env->subject`), and 25 (`subject`).

The key idea of concretization and redirection is that the concretized values are used only as keys for redirection and never actually dereferenced. Concretized pointer values ensure the same original buffer address is accessed and the accesses can simply be redirected. For example, we concretize the instruction on line 4 that reads the `header` parameter. On line 6, `pine` reads `header->env` through `header`. Since the `ebx` register is already concretized on line 4, `ecx` is also concretized on line 6. Note, however, that one instruction may execute many times, and some instances of the instruction should be concretized while other instances should not, we ensure that the instruction performs the original behavior as necessary through the else branch of the instrumentation.

4. PRACTICAL CHALLENGES

Nondeterminism. In the previous section, we claimed that the dual slice presents only differences originating from input differences. However, when a program is nondeterministic, there can also be differences caused by that nondeterminism. For example, Fig. 7 shows that two executions of `pine` that send an email can have differences as a result of nondeterminism within an event handling loop controlled by a timer. In

```
1 call_mailer(METAENV* header, ...) {
2   ...
3   smtp_mail(..., header->env, ...);
4 /* 633b98: MOV ebx, [ebp + 0x8]
5    → MOV ebx, 0x7ffee00
6    633ba0: MOV ecx, [ebx]
7    → if ebx == 0x7ffee00:
8         MOV ecx, 0x7ffef40
9       else: MOV ecx, [ebx] */
10  ...
11 }
12 smtp_mail(..., ENVELOPE* env, ...) {
13  ...
14  rfc822_output_header_line(..., env->subject);
15 /* 642182: MOV eax, [ecx + 8]
16    → if ecx + 8 == 0x7ffab40:
17         MOV eax, 0x7fff080
18      else: MOV eax, [ecx + 8] */
19  ...
20 }
21 rfc822_output_header_line(..., char * subject) {
22  ...
23  while(n-- > 0 && (**d = *subject++) != '\0')
24 /* 664fc9: MOV eax, [edx]
25    → if 0x7fff080 ≤ edx < 0x7fff080 + 10:
26         MOV eax, nSubject[edx-0x7fff080]
27      else: MOV eax, [edx] */
28  ...
29 }
```

Figure 6: Source code of **pine** reading subject from data structure

the first execution, we select the send command of the menu before the timer expires in iteration A, but in the second execution, we select the send command after the first timeout in iteration B and before a second timeout in iteration C. Thus, lines 16-20 in the first execution do not align with any lines in the second execution. The resulting dual slice includes lines 16-18 because line 19 control depends upon lines 16 and 18, and line 18 data depends upon line 17. This is not a desired result because lines 16-18 also execute in iteration C of the second execution, and these lines show no value differences.

To address this issue, our technique identifies possible nondeterminism through a calibration phase. We execute the program twice with the same input. Differences between the two executions show that the program has nondeterministic behavior, and specific differences indicate where nondeterminism occurs. When an instruction has a value difference across the executions, i.e. their occurrences in the two executions align but have different values, the value difference originates from nondeterminism and the instruction can be ignored during the component location process.

In contrast, control flow differences, i.e. unaligned instruction instances, usually arise from nondeterminism in event handling loops. In our **pine** example, the send menu item may be selected in either the first or second iteration of a loop depending on the timer. Thus, our technique first finds the loop containing the nondeterministic behavior through calibration. In our example, the **while** loop starting on line 12 is identified as the nondeterministic event handling loop.

During component identification, when aligning executions with *different* inputs, our technique does not simply align each iteration of a nondeterministic loop in order, but rather based on edit-distance[28]. The technique finds which alignment of iterations in both executions yields the fewest misaligned instructions. If multiple iterations align equally well, the earliest is selected. In Fig. 7, our technique aligns iterations A of the first execution and C of the second.

Locating Multiple Components. In some cases, the component containing the dual slice includes most of the binary. For these cases, our technique supports identifying *multiple* modular functions to extract instead of just one function, in order to reduce the size of the component. Recall, for example, the **xv** case in Fig. 5b. Locating a single component will extract everything called by the function `mainLoop()`, which is undesirable. Instead, our technique can extract only `writeJPEG()` and `LoadBMP()`, which precisely capture exactly the behavior of interest.

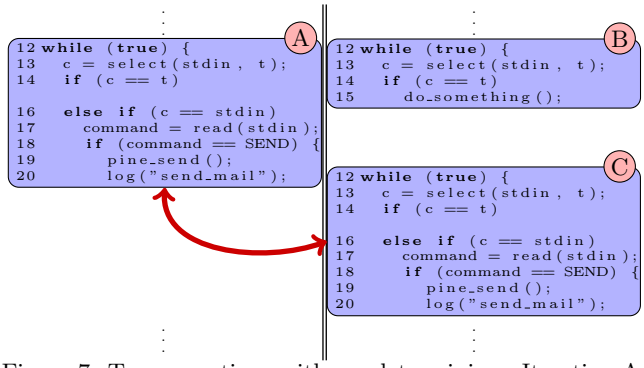


Figure 7: Two executions with nondeterminism. Iteration A should align with iteration C.

When extracting multiple functions, the data flow between the functions must be property connected. For example, `writeJPEG()` must use image data generated by `LoadBMP()`. The concretization and redirection process handles this. To simplify our discussion, assume two functions A and B are extracted with A writing to a chunk of memory m, which is later read by B. The memory is allocated outside both A and B. During concretization, instructions writing values to m (in A) are replaced with writes to a new and valid memory location. Instructions reading m (in B) are redirected to the new memory. If data flow between A and B goes through another function C, e.g. C modifies the values between A and B, C will be included in the dual slice as well, and thus the aforementioned process still applies.

Concretizing Other Resources. During concretization, accesses to external memory get replaced with valid accesses to freshly created variables. However, not all external dependences may be on memory. For example, the extracted component may depend upon a file handle that is not affected by the reuser's chosen parameters. In such cases, we must identify the external dependences on other necessary resources and safely acquire them as well. Pragmatically, our present implementation handles dependences on external files and sockets, but it can be extended to other resources once they have been identified.

5. EXPERIMENTS AND RESULTS

We have implemented a prototype of our system using Pin[30] for tracing, while the dual slicing is written in C. We use Bistro[11] for extracting the desired components as well as for binary rewriting to perform concretization and redirection. Note that Bistro is a robust binary transformation tool. It can safely extract portions from or patch arbitrary binaries by correcting internal references in the binary like indirect jump and call targets. Using our prototype, we were able to extract 10 components of interest from 7 real world programs into object files. We were further able to link with and invoke those components from new programs that reused the

extracted behavior.

5.1 Observations

Table 1 presents the 10 components we examined in our study. Although we used binaries with debugging information to clarify the dual slicing results during experimentation, our technique does not rely on any debugging symbols or information and can be applied to stripped binaries.

We first examine results for locating desired features using the coverage based approach and our dual slicing based approach. 'Cov' is the number of instructions covered by the execution exhibiting the feature and *not* covered by the execution not exhibiting the feature [41]. These locate the feature in the coverage based approach. Extracting the functions located by this approach yields the extracted component size 'Cov F. Size'. In contrast, our approach uses the dual slice to locate the desired component. 'DS' presents the number of instructions in the dual slice. Note that it is usually orders of magnitude smaller than 'Cov'. Extracting the functions identified by the dual slice yields the extracted component size 'F. Size'. This is also smaller than the size of components extracted using coverage based techniques. We note that in some cases, like Murofet, dual slicing can locate the desired component even when there are no coverage differences. Thus, dual slicing can have greater generality than a coverage based approach. Also note that existing coverage based approaches do not inherently support working on binaries or performing function extraction [41].

The 'Funcs' column is the number of the modular functions ultimately discovered with our algorithm. In one half of cases, we identified one function, but in the other half, we identified two modular functions to extract. These numbers do not include the callees of the identified functions, which are also extracted. This indicates that many features require multiple functional components. Extracting such features requires understanding relationships such as the data flow between components. Our technique automates this through concretization as presented in the previous section.

The 'C.Size' and 'C.Instrs' columns list the sizes of concretized memory and the number of concretized instructions respectively. Observe that the size of the concretized memory is relatively high compared to the number of concretized instructions. The `pine` case study in the first row shows that a concretized instruction reads roughly 32 bytes on average. This indicates that some of the concretized instructions are in loops, so they execute multiple times to read additional data. Moreover, the size of the concretized memory indicates that the extracted functions require substantial data and that extracting functions without considering these data is unlikely to work. This concretized information mostly reflects pointers through data structures and global configuration data, such as the SMTP server in `pine` example. The smaller number of concretized instructions further indicates that the size overhead caused by concretization is low.

The 'R.Instrs' column lists the number of redirected instructions. It shows that a very small number of instructions are responsible for reading the parameters of the extracted function and must be redirected for the desired interface.

For the centerim and smbc cases, extracting the feature required two components. For centerim, the login component does not have practical use. However, the send-message component relies upon the login component. To locate the two centerim components, we leveraged experience using the program. When sending a message with centerim, the program requires a password. From this, we infer that we must log into the message server to send a message. Thus, we use two inputs with different login credentials to locate the login component, and we use another two inputs with different messages to locate the send-message component. However, this knowledge is not always available. For smbc, we use two inputs different both in login information and directory name. From the resulting callgraph, we can locate both login and create-directory components.

5.2 Case Study: Murofet Worm

Many worms such as Torpig[37], Conficker[34], and Murofet use a technique called *domain flux*, which generates a list of domain names at runtime to hinder analysis of communication between a worm and the attacker. With domain flux, the defender cannot simply block IP addresses to stop the malicious behavior of the worm. Extracting the proprietary domain flux algorithm helps defend against the worm by predicting malicious domain names in advance. To further clarify the use and practicality of our technique, we have posted a demo of this Murofet case study online [4].

When Murofet executes, it generates a domain name and connects to the domain to check whether another malicious payload exists on the server. If the connection fails, the worm repeats the process until it finds a valid server.

To apply our technique, we need two executions generating domain names with different input. Since the worm does not provide a user interface for the domain flux algorithm, we cannot just change the input values. Moreover, we do not *know* the inputs for the domain flux algorithm. From multiple executions, we observe that the worm produces different domain names each time it executes, so the worm uses different inputs whenever it executes. With two executions that produce different domain names, our technique found the function responsible for generating domain names. We used the generated domain name in the DNS lookup packet as the slice criterion. From the dual slice, our technique found the modular function corresponding to the domain flux process and also found that the input for the process came from the system time by analyzing the data dependences in the slice.

This case study shows that even when a program has no user interface for providing parameters, our technique can still identify and extract the target feature. It can create an interface for parameters as long as the inputs can change across multiple executions. Once it identifies the modular component, we can also narrow our focus to that component to ease further manual analysis of the feature.

5.3 Case Study: Word97

Legacy Microsoft Word saves files in a proprietary DOC format. The format is not well documented, and reading such files is difficult to implement. Third party applications support DOC files, but they face limitations. To ensure that data in such files can be read, we consider the case of transforming a Word97 DOC file into a plain text file. With Microsoft Word, this can be achieved by opening a DOC file and directly saving it as a text file. To identify the function responsible, we use two different DOC files and save these files as different text files. Windows GUI applications have event handling loops that cause nondeterminism because we cannot control order of events. However, our calibration phase effectively identifies the events even when they occur

Program	Size	Feature	Cov	Cov F. Size	DS	F. Size	Funcs	C. Size	C. Instrs.	R. Instrs.
alpine	5.4MB	Send mail	17210	556KB	360	350KB	1	10KB	314	4
		Create a directory	5168	508KB	204	100KB	2	4KB	195	2
centerim	2.1MB	Login	21814	414KB	96	80KB	2	90KB	142	2
		Send message	11745	529KB	20	15KB	2	4KB	60	2
murofet	3.9MB	Domain flux	-	-	18	12KB	1	20KB	19	1
mysql	3.1MB	Get a list of databases	1000	116KB	270	39KB	2	5KB	187	1
ncmpc	1.2MB	Add a song to playlist	1245	160KB	65	1KB	1	500B	10	1
smbc	7.5MB	Connect to samba server	6090	407KB	1609	210KB	1	4KB	174	2
		Create a directory	6090	407KB	1609	110KB	1	300KB	354	1
word97	5.3MB	Convert DOC to text	19752	754KB	725	52KB	2	68KB	823	2
xv	2.7MB	Convert BMP to JPEG	6083	346KB	3196	50KB	2	5KB	337	5

Table 1: Extraction results

in different iterations of event handling loops.

The dynamic call tree containing the dual slice forks into two main branches. The first reads and stores the file content into memory. The second then stores this data in a text file. We thus extracted the common ancestor function within each branch into a new component that is able to read a DOC file and write it into a specified text file. Not only that, but the extracted component is self-contained and works even across different versions of the Windows platform.

This component also exemplifies the necessity of concretizing additional resources as noted in Sec. 4. Word97 creates many handles that are initialized by the kernel and that are used by the underlying components that read and write the DOC files. These external dependences on kernel objects must be valid in order for the extracted component to run, so we extended our resource concretization system to handle these kernel level resources as well in order to create the kernel objects before we execute the extracted component.

5.4 Limitations

Our technique shows that it is feasible in practice to locate and extract complex binary features into reusable software components. However, dynamic analysis imposes some limitations. In particular, we only extract a portion of the full semantics of a function. For example, the accesses observed in the provided executions are used by the technique to determine what portions of memory to concretize and make available to the extracted component. The provided executions may not access the full range of addresses in a global table. In this case, we can only guarantee that we extract the portion of the table observed in the provided executions, and accesses outside this range lead to undefined behavior.

6. RELATED WORK

Most related work within the software engineering community comes from feature location [40, 41, 8, 10, 15, 33, 9, 38, 35, 13, 6, 14, 16], which identifies where a feature is implemented within source code, and function extraction or extract method refactoring [26, 23, 17, 18, 31, 25, 10], which extracts some selected functionality out of existing source code into a self contained function. Approaches for feature location range from coverage based techniques [40, 41, 6, 14] to machine learning techniques [35]. While we showed that dual slicing can perform better than the former, the latter often require either source code or many more executions to be provided by the reuser, which is undesirable. Source code function extraction simply is not applicable to binaries.

Recent work has focused on identifying and extracting binary features of existing programs. Manual extraction [1, 2, 12] has been used to aid in understanding malware. Dynamic binary slicing [5, 47] and chopping [21] have been used for binary analysis. Execution slicing can help identify basic blocks composing a feature [42]. Chopping can help identify relationships between given inputs and outputs [27]. These work in small applications, but in complicated programs both dynamic slicing and chopping can produce a large set of instructions. Zhao et al. [48] combined static analysis and dynamic backward slicing to identify encryption and decryption logic in malware.

BCR [7] identifies interfaces and extracts functions. It identifies parameters through dynamic analysis. Compared to our technique: (1) BCR identifies all parameters, while we remove unwanted parameters to yield a concise interface. (2) BCR targets small and well defined functions, but our technique handles more complex features. As seen in Sec. 5, such features have many external dependences, and manually specifying values for each is unrealistic. BCR also does not address locating the function containing a feature.

Inspector Gadget [22] identifies and extracts a gadget with a desired feature. This work identifies gadgets using dynamic slicing, but recall that this often produces a large set of instructions. This work also focuses on the replay of the gadget and does not issues like providing an interface.

Lin et al. [29] extract and reuse functionality of a benign program within a malicious one. They identify functions by finding the common ancestor of all functions that produce the desired output. As shown in the `xv` case study, this approach may produce an undesirably large component.

7. CONCLUSIONS

We presented a novel technique for locating and extracting a binary fragment that implements a desired feature within a program. Our technique enables both an intuitive representation of the desired feature through executions and an automated location and extraction process for the feature. We also presented a technique called interface casting, which bypasses the problem of reverse engineering the complicated interface of a function and instead provides a new usable interface. Evaluation on a set of real world applications shows that our technique can extract various reusable components.

Acknowledgement

This research has been supported in part by DARPA under contract 12011593 and by NSF under awards 0845870, 0917007 and 1320326. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of DARPA and NSF.

8. REFERENCES

[1] Kraken Encryption Algorithm. http://mnin.blogspot.com/2008/04/kraken-encryption-algorithm.html, 2008.

[2] Kraken is finally cracked. http://mnin.blogspot.com/2008/04/kraken-is-finally-cracked.html, 2008.

[3] Research announcement: Binary executable transforms. Defense Advanced Research Projects Agency, 2011.

[4] Demo of murofet domain flux extraction. https://www.youtube.com/watch?v=7GU4b68oWD4, 2013.

[5] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[6] G. Antoniol and Y.-G. Gueheneuc. Feature identification: a novel approach and a case study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2005.

[7] J. Caballero, N. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[8] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. Decomposing legacy programs: a first step towards migrating to client-server platforms. In *Proceedings of the 6th IEEE International Workshop on Program Comprehension (IWPC)*, 1998.

[9] K. Chen and V. Rajlich. Ripples: Tool for change in legacy software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2001.

[10] F. Cutillo, F. Lanubile, and G. Visaggio. Extracting application domain functions from old code: a real experience. In *Proceedings of the IEEE Second Workshop on Program Comprehension (WPC)*, 1993.

[11] Z. Deng, X. Zhang, and D. Xu. BISTRO: Binary Component Extraction and Embedding for Software Security Applications. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, 2013.

[12] F. Desclaux and K. Kortchinsky. Vanilla Skype part 1. *ReCon*, 2006.

[13] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 2013.

[14] D. Edwards, S. Simmons, and N. Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 2006.

[15] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 2003.

[16] A. Eisenberg and K. De Volder. Dynamic feature traces: finding features in unfamiliar code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2005.

[17] R. Ettinger. Program sliding. In *Proceedings of the 26th European conference on Object-Oriented Programming (ECOOP)*, 2012.

[18] R. Ettinger and M. Verbaere. Untangling: a slice extraction refactoring. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, 2004.

[19] W. B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Transactions on Software Engineering*, 2005.

[20] M. N. Gagnon, S. Taylor, and A. K. Ghosh. Software protection through anti-debugging. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*, 2007.

[21] D. Jackson and E. J. Rollins. Chopping: A generalization of slicing. Technical report, SIGSOFT Symposium on the Foundations of Software Engineering (FSE), 1994.

[22] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*, 2010.

[23] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC)*, 2003.

[24] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 1988.

[25] A. Lakhotia and J.-C. Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology*, 1998.

[26] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 1997.

[27] A. Lanzi, M. I. Sharif, and W. Lee. K-tracer: A system for extracting kernel malware behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[28] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR*, 1966. English translation in Soviet Physics Doklady, 10(8).

[29] Z. Lin, X. Zhang, and D. Xu. Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010.

[30] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[31] K. Maruyama. Automated method-extraction refactoring by using block-based slicing. In *Proceedings of the 2001 Symposium on Software Reusability: putting software reuse in context (SSR)*, 2001.

[32] D. Mcilroy. Mass-produced Software Components. *Software Engineering Concepts and Techniques*, 1969.

[33] M. Petrenko and V. Rajlich. Concept location using program dependencies and information retrieval (depir). *Information and Software Technology*, 2013.

[34] P. Porras, H. Saïdi, and V. Yegneswaran. A Foray into Conficker's Logic and Rendezvous Points. In *Proceedings of the 2nd USENIX Conference on*

Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More (LEET), 2009.

[35] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 2007.

[36] C. Puttick. Preserving legacy files with ECMA Office Open XML (MSOOXML). ODF Europe Action Group, 2007.

[37] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szyd-lowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*, 2009.

[38] J. Wang, X. Peng, Z. Xing, and W. Zhao. Improving feature location practice with multi-faceted interactive exploration. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, 2013.

[39] D. Weeratunge and X. Zhang. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[40] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 1992.

[41] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 1995.

[42] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 2000.

[43] W. E. Wong, J. R. Horgan, S. S. Gokhale, and K. S. Trivedi. Locating program features using execution slices. In *Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET)*, 1999.

[44] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[45] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[46] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[47] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, 2003.

[48] Z. Zhao, G.-J. Ahn, and H. Hu. Automatic extraction of secrets from malware. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE)*, 2011.